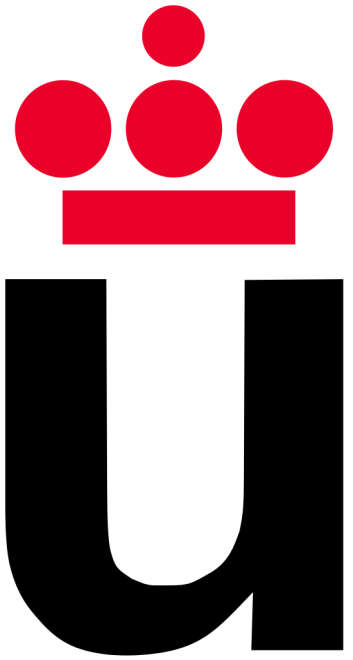
Desarrollo de juegos con inteligencia artificial

Práctica 1



Luis Antonio González Martínez
Sergio Jesús González Castilla
19/11/2023

ÍNDICE:

1. Introducción: Contexto de la práctica

Para el desarrollo de la práctica será necesario su contextualización, el objetivo de la misma será implementar un agente inteligente (en este caso, nuestro personaje principal del entorno), además del algoritmo de búsqueda necesario para que sea capaz de desplazarse por nuestro entorno virtual. Para resolver este problema, se utilizará uno de los motores de videojuegos más comunes de nuestro sector “Unity”, y su lenguaje de programación “C#”.

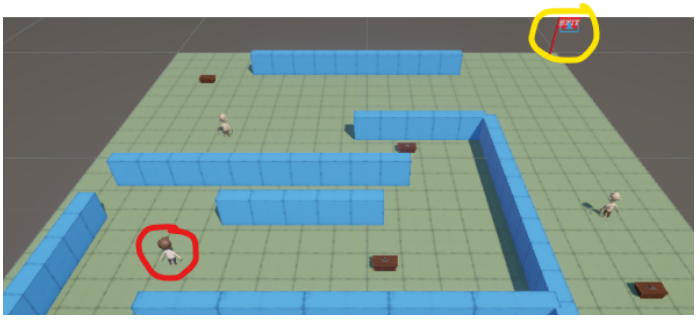


Figura 1: Agente inteligente y estado meta

2. Problema que se le plantea a la IA

El problema planteado a resolver no es ni más ni menos que utilizar un algoritmo de búsqueda para conseguir que nuestro agente inteligente encuentre su objetivo dentro de un entorno, Deberá de atravesar todo el mapa desde un estado inicial, hasta el estado meta (el objetivo principal de la práctica). Para ello, se necesita que el algoritmo escogido llegue al nodo meta de la forma más óptima, en otras palabras, que sea capaz de calcular desde el primer momento el recorrido más corto desde la posición inicial del personaje hasta su resolución. Lo ideal sería omitir estados ya visitados anteriormente (nodos repetidos) para reducir el gasto computacional.

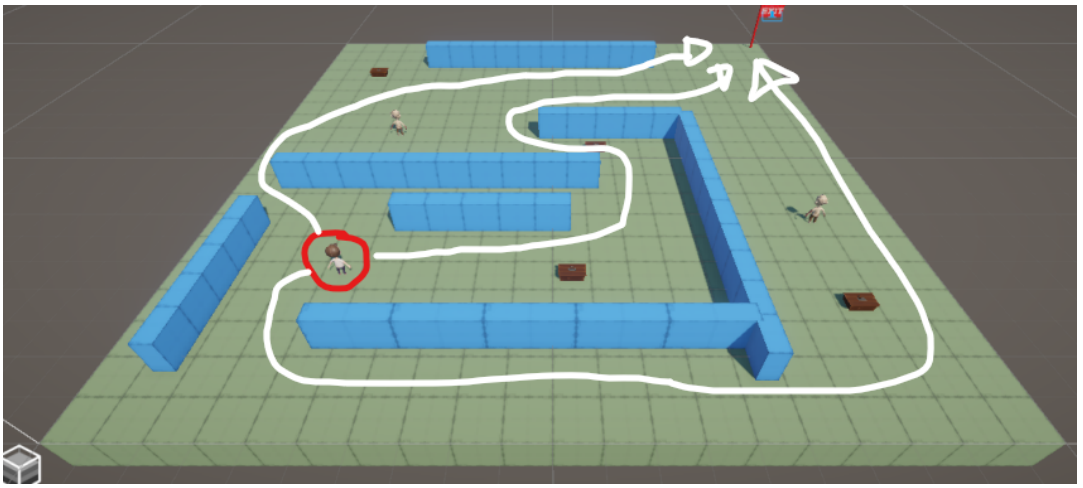


Figura 2: Agente inteligente y estado meta

3. Algoritmos implementados

Tras el planteamiento del problema y su posible solución, se debe plantear el algoritmo que utilizaremos para el desempeño de su óptima resolución. Estudiando la situación, se considera que la mejor solución al problema lo puede aportar el algoritmo A*.

El mismo se asemeja a una búsqueda inteligente o informada, que se encarga de hallar el camino más corto por el que puede ir nuestro agente inteligente. La ruta iniciará en un estado inicial y se recorrerá hasta un estado meta, usando siempre una heurística óptima, o lo que es igual, definirá el camino más corto hasta el destino.

Búsqueda A*

```
1:  procedure SPACESTATESEARCH(initialState)
2:      openList ← {initialState}
3:      while openList is not empty do
4:          currentState ← first(openList)
5:          if currentState is_goal? then
6:              return currentState
7:          end if
8:          successorStates ← expand currentState
9:          for all successor in successorStates do
10:             succesor.parent ← {currentState}
11:             openList ← {succesor}, succesor.f
12:         end for
13:     end while
14:     return no solution found
15: end procedure
```

Figura 3: Búsqueda A*

Este algoritmo usa la búsqueda basada en heurística, es decir, utiliza una función que consta de la suma del coste de llegada al nodo (coste g) y el posible coste de llegada al nodo final (coste h*). Posteriormente, al elegir el nodo que se deba expandir se elige el nodo cuyo resultado de la función heurística sea menor. Este tipo de algoritmo funcionará muy bien cuando no existan estados infinitos en el entorno donde coloquemos nuestro agente inteligente, y al ser siempre admisible, es una muy buena opción a implementar dentro de nuestro proyecto.

4. Controles de los algoritmos

En el apartado 5 se explicará con precisión el funcionamiento del código desarrollado, en este se especificará cómo se maneja el programa Unity para ejecutar y seleccionar el algoritmo y el agente. Para poder ejecutar el código y ver que funcione, será de obligación tener Unity descargado en el ordenador, en específico la versión 2022.3.2.f1, además deben instalarse los packages que se emplean en el manager de Unity “Probuilder y CineMachine”.

Por último, en cuanto a ajustes del programa, se debe instalar el package Unity facilitado en esta nuestra práctica, que serán la solución correspondiente al ejercicio 1 y 2 de la misma, esta se descargará a través de la carpeta compartida de la práctica por el Grupo A en la entrega de la misma. Se debe asegurar que se acepte cada uno de los elementos del paquete antes de proceder.

Una vez implementado el proyecto, se debe de abrir la escena del mismo para así poder probar las funcionalidades, el agente (controlado por el algoritmo) y el entorno (el escenario completo con todo sus elementos). Seleccionado al Agente Inteligente, se debe de seleccionar en las opciones del mismo, tanto el agente inteligente utilizado como el algoritmo empleado. Para ejecutar el código y ver el resultado, debe de pulsar el botón superior “play” y así poder ver la resolución de nuestra práctica.

5. Código comentado

Para finalizar, encontrarás el código con un comentario sobre la funcionalidad del mismo, para así tener claro cuál es su objetivo en ejecución.

5.1. Clase Node

El siguiente fragmento de código corresponde a nuestra clase “Nodo”. Esta clase representa los posibles estados en los que nuestro agente se moverá una vez implementado en nuestro algoritmo A*.

En primer lugar, definimos la clase e implementamos las interfaces "IComparable<Node>" e "IEquatable<Node>". Estas implementaciones permiten comparar instancias de la clase utilizando esos métodos.

```
namespace Assets.Scripts.GrupoA.AStar
{
    public class Node : IComparable<Node>, IEquatable<Node>
    {
        public CellInfo info;
        public Node padre;
        public float G_Coste;
        public float H_Heuristica;
        public float Funcion_Heuristica;
    }
    ...
}
```

Dentro del nodo nos encontramos lo siguiente :

- info : Contiene información de celdas del mundo.
- padre: Representa el nodo padre/procedente.
- G_Coste: Representa el coste de búsqueda con respecto al nodo de inicio.
- H_Heuristica: Representa el valor de la heurística, desde el nodo hasta el nodo meta. En caso de que se usase una búsqueda voraz, ya se tiene la variable disponible.
- Funcion_Heuristica : Representa la suma de las dos variables anteriores.

Después se inicializa el constructor, el cual crea nodo con instancias de la anterior información.

```
    public Node(CellInfo info, float G_Coste = 0, Node padre = null)
    {
        this.info = info;
        this.padre = padre;
        this.G_Coste = G_Coste + 1;
        this.H_Heuristica = 0;
        this.Funcion_Heuristica = 0;
    }
```

A continuación se calcula la distancia Manhattan, para así calcular la heurística entre el nodo y el nodo meta. “calculateHeuristic” se encarga de calcular el valor heurístico basado en la distancia manhattan y actualiza el valor de “H_heuristica” y “Funcion_Heuristica”.

```
public float D_Manhattan(Node targetNode)
{
    return (Math.Abs(targetNode.info.x - this.info.x) +
            Math.Abs(targetNode.info.y - this.info.y));
}
public float calculateHeuristic(Node targetNode)
{
    H_Heuristica = D_Manhattan(targetNode);
    Funcion_Heuristica = G_Coste + H_Heuristica;
    return Funcion_Heuristica;
}
```

Para finalizar se compara y se iguala;

CompareTo: Compara dos nodos basados en sus valores de Función Heurística.
Equals: Comprueba si dos nodos son iguales basados en su Función Heurística.
EqualsNode: Comprueba si dos nodos son iguales según su información (estados).

```
public int CompareTo(Node other)
{
    if (other == null) return 1;
    return this.Funcion_Heuristica.CompareTo(other.Funcion_Heuristica);
}
public bool Equals(Node other)
{
    if (other == null) return false;
    return this.Funcion_Heuristica == other.Funcion_Heuristica;
}
public bool EqualsNode(Node other)
{
    return this.info.x == other.info.x && this.info.y == other.info.y;
}
```

5.2. Algoritmo A Star

El siguiente fragmento de código corresponde a la implementación del algoritmo A*. En la práctica, la elección de este algoritmo se basó en su capacidad para encontrar la solución más rápida, es decir, el camino más corto. En una primera instancia, se implementa la interfaz “INavigationAlgorithm”, la cual se utiliza como un estándar para algoritmos de búsqueda en el sistema desarrollado.

```
namespace Assets.Scripts.GrupoA.AStar
{
    public class AStar : INavigationAlgorithm
    {
        private WorldInfo _world;
        private List<Node> openList = new List<Node>();
        private List<Node> closeList = new List<Node>();
    }
    ...
}
```

El algoritmo inicializa dos listas:

- openList, siendo esta la lista de nodos candidatos a ser los más óptimos para ser explorados en la resolución.
- closeList, que es una lista que almacenará los nodos por los que ya haya pasado nuestro agente racional.

En el fragmento de código siguiente, se introduce la variable ”_world” en el constructor, la cual representa la información del entorno o mundo en el que se llevará a cabo la búsqueda.

```
    public void Initialize(WorldInfo worldInfo, INavigationAlgorithm.
AllowedMovements allowedMovements)
    {
        _world = worldInfo;
    }
```

Continuando, nos encontramos con los métodos GetPath, método heredado de INavigationAlgorithm que recibe las celdas iniciales y finales. Llama al método GetNodePath (explicado más adelante) y convierte el historial de los nodos padres del nodo final recibido (si es que tiene) en una lista de los nodos por lo que debe pasar el agente

```
    public CellInfo[] GetPath(CellInfo startNode, CellInfo targetNode)
    {
        Node current = GetNodePath(startNode, targetNode);
        List<CellInfo> temporal = new List<CellInfo>();
```

```
            while (current != null)
            {
                temporal.Add(current.info);
                current = current.padre;
            }
            temporal.Reverse();
            return temporal.ToArray();
        }
```

A su vez, GetPath llama a la función GetNodePath que le pasa la celda inicial y objetivo. Aplica el algoritmo estrella buscando la mejor ruta, y devuelve el nodo con sus padres.

```
        public Node GetNodePath(CellInfo startNode, CellInfo targetNode)
        {
            openList.Clear();
            closeList.Clear();
            Node current = new Node(startNode);
            Node target = new Node(targetNode);
            openList.Add(current);
            while (openList.Count > 0)
            {
                current = openList.First();
                openList.RemoveAt(0);
                closeList.Add(current);
                if (current.info.x == target.info.x &&
                    current.info.y == target.info.y)
                {
                    return current;
                }
                else
                {
                    Node[] sucesors = Expand(current, target);
                    foreach (Node sucesor in successors)
                    {
                        if (!visited(sucesor))
                        {
                            openList.Add(sucesor);
                        }
                    }
                    openList.Sort();
                }
            }
            return null;
        }
```

Por otra parte, GetNodePath, usa los métodos “Expand” y “visited”.

Anteriormente, se inicializaron las listas "abierta" y "cerrada", añadiendo nodos a la lista. El bucle no se detiene hasta que ambas listas están llenas o se haya encontrado el nodo meta. En cada iteración, se selecciona el nodo con el menor coste de la lista "abierta", se elimina de esa lista y se agrega a la lista "cerrada". Si el nodo es el destino, se devuelve como resultado. En caso contrario, el nodo se expandirá y se agregan los nodos resultantes a la lista "abierta" si aún no han sido visitados previamente. Al finalizar, la lista se ordenará (.sort) según el coste para llegar al nodo meta.

```
    public bool visited(Node node)
    {
        foreach (Node lista in closeList)
        {
            if (lista.EqualsNode(node))
            {
                return true;
            }
        }
        return false;
    }
}
```

La orden de expansión recibe el nodo padre y el nodo objetivo. Usando la variable _world, explora las casillas cercanas y comprueba si son “walkable”, es decir, comprueba si el agente puede pasar por ellas. Si es afirmativo los añade en el array y posteriormente se usa el objetivo para calcular las heurísticas.

```
    public Node[] Expand(Node padre, Node targetNode)
    {
        List<Node> nodes = new List<Node>();
        CellInfo[] cells = {
            _world[padre.info.x, padre.info.y+1],
            _world[padre.info.x+1, padre.info.y],
            _world[padre.info.x, padre.info.y-1],
            _world[padre.info.x-1, padre.info.y]
        };
        if (cells[0].Walkable)
        {
            nodes.Add(new Node(cells[0], padre.G_Coste, padre));
        }
        if (cells[1].Walkable)
        {
            nodes.Add(new Node(cells[1], padre.G_Coste, padre));
        }
```

```
            if (cells[2].Walkable)
            {
                nodes.Add(new Node(cells[2], padre.G_Coste, padre));
            }
            if (cells[3].Walkable)
            {
                nodes.Add(new Node(cells[3], padre.G_Coste, padre));
            }
            nodes.ForEach(node => { node.calculateHeuristic(targetNode); });
            return nodes.ToArray();
        }
}
```

5.3. Subobjetivos

En este código se define la clase de subobjetivos que implementa INavigationAgent.

Primero se define las propiedades:

- CurrentObjective : representa el objetivo destino del agente
- CurrentDestination: representa el destino en coordenadas del mundo
- NumberOfDestination: Representa el total de destinos del Agente

```
public class SobobjectiveSearchAgent : INavigationAgent
    {
        public CellInfo CurrentObjective { get; private set; }
        public Vector3 CurrentDestination { get; private set; }
        public int NumberOfDestinations { get; private set; }
```

Sabiendo que tenemos:

- worldInfo: información del mundo
- _navigationAlgorithm: instancia del algoritmo que usa el agente
- origin: representa la posición inicial del agente
- objectives: un array de cell con información de los destinos
- path: es la cola que el agente tiene que seguir.

```
        private WorldInfo _worldInfo;
        private INavigationAlgorithm _navigationAlgorithm;

        private CellInfo _origin;
        private CellInfo[] _objectives;
        private Queue<CellInfo> _path;
```

Trás saber esto, se inicializa un constructor con el agente y la información del mundo con la posición inicial. Además de inicializar el algoritmo con la información del mundo.

```
public SobobjectiveSearchAgent(WorldInfo worldInfo, INavigationAlgorithm
       navigationAlgorithm, CellInfo startPosition)
{
    _worldInfo = worldInfo;
    _navigationAlgorithm = navigationAlgorithm;
    _origin = startPosition;

    _navigationAlgorithm.Initialize
    (_worldInfo, INavigationAlgorithm.AllowedMovements.FourDirections);
}
```

En el método de “GetNextDestination”, se verifica si los objetivos están definidos, sino se inicializan. Si ya se ha completado y llegado al primer objetivo, se crea un nuevo “path” para la nueva posición a recorrer. Se irá actualizando el destino y lo irá devolviendo.

```
public Vector3? GetNextDestination()
{
    if (_objectives == null)
    {
        _objectives = GetDestinations();
        CurrentObjective = _objectives[_objectives.Length - 1];
        NumberOfDestinations = _objectives.Length;
    }

    if (_path == null || _path.Count == 0)
    {
        if (NumberOfDestinations > 0)
        {
            NumberOfDestinations--;
            CurrentObjective = _objectives[NumberOfDestinations];
        }
        CellInfo[] path = _navigationAlgorithm.GetPath
        (_origin, CurrentObjective);
        _path = new Queue<CellInfo>(path);
    }

    if (_path.Count > 0)
    {
        CellInfo destination = _path.Dequeue();
        _origin = destination;
        CurrentDestination = _worldInfo.ToWorldPosition(destination);
    }
```

```
            return CurrentDestination;
        }
```

El método "GetDestinations" inicia creando una lista de nodos que representan los objetivos en el mundo. Estos nodos se ordenan según su valor heurístico en relación con la posición actual del agente. Luego, se agrega el nodo de salida a la lista y se vuelve a ordenar. Finalmente, la lista de nodos se convierte en una lista de objetos "CellInfo" y se devuelve como un array.

```
        private CellInfo[] GetDestinations()
        {
            List<Node> targets = new List<Node>();
            CellInfo[] objetivos = _worldInfo.Targets;
            Node exit = new Node(_worldInfo.Exit);
            foreach (var item in objetivos)
            {
                Node tmp = new Node(item);
                //tmp.calculateHeuristic(exit);
                tmp.calculateHeuristic(new Node(_origin));
                targets.Add(tmp);
            }
            targets.Sort();
            targets.Add(exit);
            targets.Reverse();
            List<CellInfo> destinations = new List<CellInfo>();
            foreach (var item in targets) { destinations.Add(item.info); }
            return destinations.ToArray();
        }
    }
}
```

6. Bibliografía

David María Arribas y Dan Casas Guix (Año académico 2023/24).
Búsqueda con heurísticas débiles [presentación].

Laura Llopis Ibor (Año académico 2023/24).
C# cheat sheet [presentación]